JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Model-Driven Development and Container Orchestration for Self-Adaptive Cloud Applications

Mufasir Muthaher Mohammed[*], **Karim Jahed**[*], **Reza Ahmadi**[†], **Juergen Dingel**[*], **and David Lamb**[*]
[*]School of Computing, Queen's University, Canada
[†]École de Technologie Supérieure, Université du Québec, Canada

ABSTRACT Containers are self-contained units of code that can be efficiently executed in various computing environments. Container orchestration tools assist in deploying, scaling, and managing containers, permitting alterations to the execution platform (environment) at runtime. Orchestration tools such as Kubernetes have become indispensable for large-scale, industrial distributed systems. Container orchestration and model-driven engineering (MDE) both offer concepts, techniques, and tools that facilitate the realization of self-adaptation capabilities. Yet, their *joint* use for the design, implementation, and maintenance of adaptive cloud applications appears to be underexplored.

This paper presents the results of our work to address this shortcoming by investigating how container orchestration can augment MDE techniques for the effective design, implementation, and maintenance of adaptive cloud applications. We will describe an approach and toolchain for automatically generating and deploying a fully containerized distributed application from a component-and-connector model and leveraging both model- and platform-level dynamic adaptation and failure recovery capabilities to allow the application to respond to changes to the requirements or failures at runtime. The application of the approach with the help of a prototype implementation of our toolchain to an exemplar will be described. Evaluation results show the feasibility and effectiveness of the approach.

KEYWORDS Model-Driven Development, Containerization, Orchestration, Self-Adaptation, and Cloud Applications.

1. Introduction

The industrial interest in self-adapting software systems is on the rise. Anecdotal evidence for this claim can be found on the web, but stronger support can also be found in the literature (e.g., (Beyer et al. 2016), (Spyker 2020), (Weyns et al. 2023)). A recent survey (Weyns et al. 2023) finds that large parts of industry already make significant use of self-adaptation to, e.g., increase system utility and decrease costs via auto-scaling, auto-tuning, or monitoring. A lot of this use is enabled by cloud computing in general and *containerization* in particular. E.g., Kubernetes is the technology most frequently mentioned in (Weyns et al. 2023), followed by AWS Elastic Cloud, RedHat OpenShift, and DynaTrace.

Industry appears positive about self-adaptation, but also reports on significant challenges and calls for further research to help alleviate them. Many of these challenges are caused by, e.g., a lack of design guidelines, the need to support different system views, and increasing complexity (Weyns et al. 2023). In general, concepts, techniques, and tools from software architecture and MDE seem particularly relevant to study and address these challenges.

Our work advocates a more comprehensive study of adaptation mechanisms and aims to integrate adaptation capabilities involving different artifacts and technologies along an MDE-for-cloud-computing toolchain. Between the model-level and the execution platform, we can identify at least four different kinds of artifacts that adaptation can target:

JOT reference format:
Mufasir Muthaher Mohammed, Karim Jahed, Reza Ahmadi, Juergen Dingel, and David Lamb. *Model-Driven Development and Container Orchestration for Self-Adaptive Cloud Applications*. Journal of Object Technology. Vol. vv, No. nn, yyyy.
http://dx.doi.org/10.5381/jot.yyyy.vv.nn.aa

1. *Model-level descriptions of system behavior:* An example of the description of the behavior of parts of the system being changed to allow the system to adapt is parameter adaptation: Behaviour is described in terms of parameters and thresholds the modification of which affects execution. Parameter adaptation is one of the most frequently used adaptation mechanisms. Another, more general, example would be the modification of programs or state machine models at runtime, as advocated by approaches to live programming and modeling (McDirmid 2016; van Rozen & van der Storm 2019; Bagherzadeh et al. 2021).
2. *Model-level descriptions of system structure:* But, adaptation can be achieved at an architectural level by dynamic changes to the specification of the structure of the system through, e.g., the addition, removal, or modification of components or connectors. Often, these kinds of changes are realized through dedicated constructs in an architecture description language (ADL) such as π-ADL (Oquendo 2004), Rainbow (Garlan et al. 2004), Stitch (Cheng & Garlan 2012), or UML-RT (Kahani et al. 2017).
3. *Platform-level descriptions of computing system topology:* The ability to make changes to computing nodes and redeploy them dynamically is one way to achieve adaptations at the platform-level. Container orchestration platforms such as Kubernetes allow runtime changes to, e.g., the number of computing nodes and the way software components are assigned to these nodes. Dynamic redeployment can help reduce communication latency and improve system responsiveness.
4. *Platform-level descriptions of computing system resources:* Container orchestration platforms allow changes to CPU and memory resources at runtime. These changes can improve system performance and responsiveness while minimizing costs incurred by allocated, yet unused resources.

MDE appears well-suited to help organize and integrate diverse, heterogeneous adaptation capabilities that reside on different levels and have different properties. In this paper, we describe our work on facilitating the design and implementation of self-adaptive, containerized cloud applications through MDE, component-and-connector (C&C) architectures, the actor model, and the effective use of model- and platform-level adaptation capabilities. Our main contributions are an approach for the model-driven development of such applications, together with a prototype implementation and its evaluation. The implementation consists of a toolchain that takes an executable C&C model of the application as input and generates and deploys a Kubernetes-based, distributed cloud application capable of autonomously recovering from node failures and adapting to changing requirements through runtime modifications to the model or the Kubernetes platform.

The paper is structured as follows: After reviewing background and related work (Section 2), we formulate the research questions our work attempts to answer (Section 3). Section 4 describes an exemplar, i.e., an example of a more complex software application that would benefit from the adaptation capabilities our work targets. Section 5 describes our approach in general terms, as well as how it has been instantiated and realized in our prototype implementation. Section 6 provides a runtime view of the cloud environment used for our implementation. Section 7 discusses how we have evaluated our approach and prototype, presents evaluation results and artifacts, and provides answers to the research questions. Section 8 discusses some more general aspects of the work. Section 9 concludes and presents future work.

2. Background and Related Work

2.1. Cloud Computing, Containerization, and Container Orchestration

Cloud computing offers on-demand availability of system resources such as data storage or computing power. The infrastructure hosting these resources does not need to be owned or managed by the user, is located off-site and accessed via the Internet. Cloud computing can help increase availability, agility, and scalability through redundancy and on-demand provisioning. Cloud providers can give customers varying degrees of access and control of the underlying infrastructure. E.g., in Infrastructure as a Service (IaaS), customers can control low-level infrastructure details such as location, scaling, and backup. In Software as a Service (SaaS) on the other hand, customers only access application software while providers manage the resources. The Platform as a Service (PaaS) model provides a middle ground between IaaS and SaaS.

Containerization facilitates the creation of software component packages that can be deployed with ease and can run on any node, irrespective of the environment, as long as the container runtime is accessible. This allows a deployment framework to redeploy components in a way that improves failure recovery and resource utilization.

Container orchestration (Casalicchio 2019) refers to the automated management of containerized applications. Orchestration platforms such as Kubernetes facilitate, e.g., provisioning, deploying, scaling, and networking of containers across multiple nodes. Orchestration platform services typically span the IaaS and PaaS parts of the service model spectrum. Containerization and orchestration offer diverse adaptation capabilities, including redeployment, platform modification, and resource management.

In sum, our work aims to offer SaaS-type encapsulation of cloud platform resources and use container orchestration to automatically ensure their effective use.

2.2. Cloud-Native Adaptation

Cloud-native applications leverage cloud platforms for elasticity, load balancing, and on-demand resource provisioning. *Elasticity* (Mell & Grance 2011; Herbst et al. 2013) refers to the ability of the system to dynamically adapt by provisioning or de-provisioning resources in response to changes in workload, aiming to closely match available resources to the present demand at any specific point in time. This elasticity gives rise to enhancing resource efficiency as well, by optimizing performance under variable workloads, scaling up for peak loads and down during reduced activity.

A survey (Hummaida et al. 2016) on adapting computation and storage resources identifies the following three prerequisites to realizing effective adaptation: A good understanding of the type of workloads, accurate profiling of these workloads in real-time, and scalable adaptation mechanisms to ensure that they can expand in tandem with the growth of the cloud system.

Authors of (Farokhi et al. 2015) suggest that relying solely on elasticity features offered by cloud service providers is insufficient, as these depend on users' knowledge to configure the elasticity mechanism. To address this, the authors suggest leveraging autonomic computing approaches based on control theory, to handle dynamic changes at runtime.

2.3. Modeling and MDE for Adaptive Systems

The use of modeling in the context of adaptive systems is broad (Weyns 2020) and includes architecture description languages (Kramer & Magee 2007; Garlan et al. 2004; Kahani et al. 2017); metamodeling, domain-specific languages (DSLs) (Alfonso et al. 2021; Vogel & Giese 2014); model transformation, model analysis, formal specification and verification (Bradbury et al. 2004); models at runtime (Bencomo et al. 2019); modeling of requirements, variability, uncertainty, features, goals, and performance, model-based testing (Pueschel et al. 2013); and design space exploration (Saxena et al. 2010). In accordance with the taxonomy dimensions identified in (Salehie & Tahvildari 2009; Krupitzer et al. 2015), the following MDE-based adaptation approaches appear most closely related to our work.

EUREMA (Vogel & Giese 2014) is based on executable runtime megamodels for developing adaptation engines through the design, execution, and adaptation of feedback loops. Relevant aspects of the software and its environment are monitored and captured using runtime models (Bencomo et al. 2019). Apart from executable runtime models, EUREMA uses DSL and megamodeling to provide an integrated view of several models and their relationships. Rainbow (Garlan et al. 2004) is an architecture-based approach with a focus on reusability for creating self-adaptive systems. Architectural models represent a system's architecture in terms of components, layers, features, implementations, interfaces, and connectors. They aid in identifying the best system configuration for improved performance and customization. Rainbow's key features are its clearly defined architecture, behavioral strategies, and utility preferences. PLASMA (Tajalli et al. 2010) adapts to changing requirements by generating plans based on user-provided goals and component specifications. It is flexible in defining adaptation requirements and covering self-configuring properties. FUSION (Elkhodary et al. 2010) is a feature-oriented self-adaptive system that aims to find a different set of features to meet goals in case of violations. The adaptation cycle modifies managed resources by turning the features on and off. DCL (Nakagawa et al. 2012) uses control loops to collect and analyze data, make decisions, and take action. These loops make up the adaptation logic, which can be dynamically modified. Finally, there are approaches in which adaptation is driven by non-functional requirements such as reliability and performance. E.g., KAMI (Epifani et al. 2009) features an adaptation logic that continuously checks to what degree non-functional requirements are met and performs adaptation to avoid potential violations.

In sum, adaptation in the reviewed approaches (Vogel & Giese 2014; Elkhodary et al. 2010; Nakagawa et al. 2012; Epifani et al. 2009; Garlan et al. 2004; Tajalli et al. 2010) is triggered by changes in the system context, failures, or user requirements. Parameter adaptation is primarily used to modify system behavior and self-configuration is the most studied self-property for adaptation, often triggered by constraint violation.

3. Research Questions

Our work differs from existing research in several ways. First, our approach aims to integrate adaptation at different levels: at the model-level by changing the way the structure and behavior of the system are described in the model, and at the platform-level through dynamic redeployment, topology changes, and resource configuration enabled by existing container orchestration techniques and tools. Second, our approach leverages existing container failure recovery techniques to make applications more resilient. Third, our work targets distributed, cloud-native applications with stateful behavior. To the best of our knowledge, none of the existing approaches offer this combination of features.

We note that a first, high-level sketch of the approach presented here was discussed at the 2022 MODELS Doctoral Symposium (Mohammed 2022). The paper also discusses the UAV exemplar. However, implementation of the approach and the exemplar was still in progress at the time. Consequently, the doctoral symposium paper does not contain any evaluation results. In contrast, the current submission contains detailed descriptions of a refined version of the approach, a complete supporting toolchain, the exemplar, as well as full evaluation details.

To address the gaps in the literature described above, our work aims to answer the following research questions:

- **RQ1:** Is it possible to adapt an existing MDE approach and toolchain to obtain an MDE approach and toolchain that allows the model-driven development of containerized applications deployable on local and remote clusters?
- **RQ2:** Is it possible to further adapt this MDE approach and toolchain for containerized applications so that the failure recovery capabilities of existing container management platforms are leveraged?
- **RQ3:** Is it possible to extend this MDE approach and toolchain to support dynamic changes to the application requirements via adaptation on a) the model-level and b) the platform-level?

4. Exemplar

We have evaluated our approach using a case study inspired by DARTSim (Moreno et al. 2019). An overview of the case study is given in Figure 1. The objective is to simulate a team of unmanned aerial vehicles (UAVs) flying a pre-determined path at a constant speed in a hostile environment, detecting as many targets as possible while avoiding being shot down by threats. The path is divided into segments of equal length,

and the sensors report if they detect a target or a threat in each segment. The UAVs use forward-looking sensors to detect targets and threats, constructing a probability distribution for each route segment ahead based on multiple observations. The UAVs can use a downward-looking sensor to detect targets on the ground but must balance the proximity to the ground with the risk of being destroyed by threats.

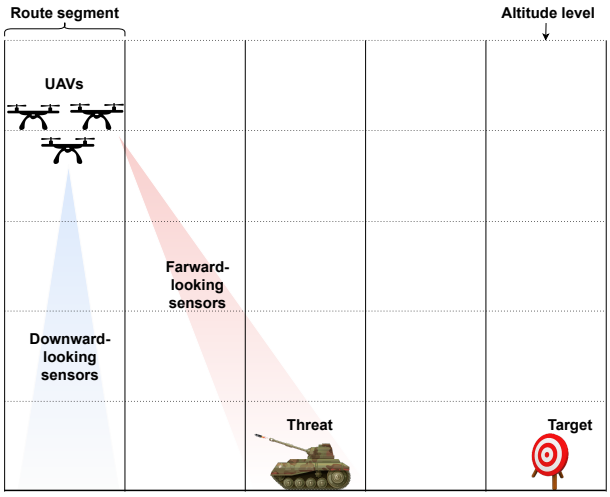

Figure 1 Overview of the Simulation

The UAVs can fly at different altitudes and in either a loose or tight formation. Altitude and formation impact the UAVs' ability to detect targets (e.g., a tight formation increases sensor overlap which impedes target detection), but also their probability of being shot down. The UAVs can also use electronic countermeasures (ECM) to decrease the probability of being destroyed by threats, but this also reduces the probability of detecting targets.

Different tradeoffs between mission outcomes allow the UAVs to be used for different kinds of missions, including *surveillance* (in which survival of UAVs is maximized, possibly at the expense of the number of targets found), *attack* (in which the number of found targets is to be maximized, even if UAVs are more likely to be hit), and compromises between these two. To be able to evaluate the suitability of our approach to runtime changes to high-level end-user requirements, runtime changes to these mission types should be possible.

We also need to be able to determine the effectiveness of adaptations, i.e., how adaptation actions impact the system performance with respect to these missions. We will use the following metrics to evaluate the performance of different sets of simulation runs:
1. Average destruction position (ADP): The average distance the UAVs were able to cover. Values lie between 0 and 30.
2. Average number of targets found (ANTF): The average number of targets the UAVs were able to find before the end of the simulation (i.e., before reaching the end of the path or being shot down).
3. Average mission success factor (AMSF): The mission is deemed successful if the UAVs manage to detect at least half of the targets without being destroyed by threats. The mission success factor is the ratio of successful missions (e.g., 3 out of 4 successful missions would give a success factor of 0.75).

To evaluate the system, multiple simulation runs are performed. The above metrics are computed as the averages across these multiple runs.

4.1. Strategies

Strategies are a model-level concept for supporting and realizing different mission types. More precisely, a strategy corresponds to a particular setting of some of the parameters that influence the behavior of the UAVs. We will consider the following three parameters: the altitude at which the UAVs fly, the formation that they fly in, and whether or not ECM is used. For each of these parameters, only two values will be considered: altitude (low, high), formation (loose, tight), and ECM (on, off). For instance, flying the UAVs at high altitudes, in a tight formation, and with ECM turned on gives rise to a *conservative strategy* suitable for surveillance-type missions where the long-term survival of the UAVs is paramount. For attack-type missions, an *aggressive strategy* can be used in which UAVs fly low, in loose formation, and with ECM turned off. A *balanced strategy* sits between these two extremes. More strategies are possible, but our implementation currently focuses on these three. To support runtime changes to the mission type, our exemplar allows for strategies to be changed at runtime.

4.2. Comparison of our Simulation with DARTSim

Our simulation differs significantly from the initial implementation of DARTSim in (Moreno et al. 2019). Most importantly, the original DARTSim was a monolithic, non-distributed C++ application that did not use containerization or any other cloud computing technology. Moreover, it was not developed using MDE and it did not support strategies that could change at runtime.

In contrast, our version is developed using MDE techniques and tools. Using our KubeRT (KubeRT 2021) toolchain, the model is automatically partitioned, containerized, and deployed on a distributed, multi-node cloud-native environment using Google Cloud Platform (GCP), Docker, and Kubernetes. Moreover, our model offers additional support for adaptation through, e.g., strategies that can be changed at runtime and a more comprehensive monitoring infrastructure.

5. Approach and Prototype Toolchain

Our approach consists of the following 5 steps (Figure 2):

1. Modeling of the system.
2. Identification of relevant runtime information and modeling of suitable monitors.
3. Use of model transformation to integrate monitors and to prepare the model for containerization.
4. Partition model and generate code and scripts.
5. Containerize and deploy.

Each step is described in more detail below.

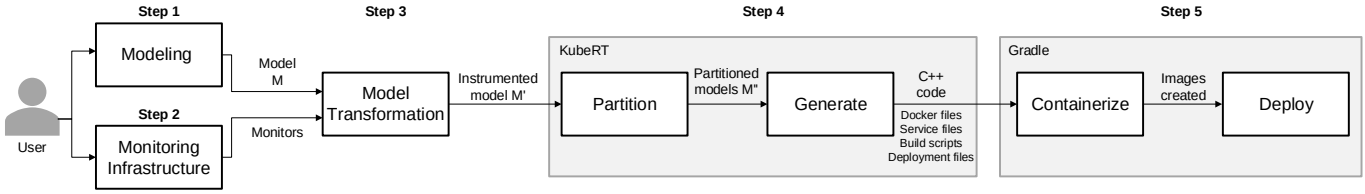

Figure 2 Overview of the Approach Supporting Tool Pipeline

5.1. Step 1: Modeling the System

In the modeling step, the user develops a model of the application. In our prototype implementation, we use a VS Code extension[1] for a textual language based on UML-RT (RTPoet-DSL 2021). UML-RT is a mature language (Selic et al. 1994; Selic 2022) with good tool support (e.g., IBM RSA-RTE, HCL RTist[2], Protos ETrice[3], Eclipse Papyrus-RT[4]) that has proven its utility for the model-driven development of reactive, state-based software as found in real-time embedded and cyber-physical systems. However, other languages and tools could be used as long as they support

- the description of executable component-and-connector (C&C) architectures using the actor model (Agha 1985),
- the collection of necessary model-level runtime information (Step 2) and model transformation (Step 3),
- the generation of executable code for the modeled system that can then be fed into subsequent steps for partitioning, generation, containerization, and deployment (Steps 4 and 5), and
- the runtime interaction with the cloud environment that the application will be deployed to collect runtime information about the platform (e.g., utilization) and to trigger changes to this platform through dynamic redeployment.

While the use of other languages and tools may necessitate customization or even reimplementation of parts of our prototype, the satisfaction of the above requirements will help reduce size and complexity of these changes. A more detailed discussion of the pros and cons of UML-RT for system description will be given below (Section 8).

Our approach assumes that the model provided by the user supports adaptation. More concretely, we suggest to concentrate the selection of suitable adaptations based on available runtime information in an Adaptation Manager component. The Adaptation Manager should collect the runtime information from the monitors (Step 2) and be able to trigger suitable model-level or platform-level adaptations. The details of the design of the Adaptation Manager and the system on the whole, is up to the user and our approach makes no further assumptions about them other than to observe that the use of the MAPE-K reference architecture (Kephart & Chess 2003) or suitable variants (Porter et al. 2020) to structure is recommended.

[1] Available at https://github.com/kjahed/kubert-vscode-extension
[2] Available at https://github.com/HCL-TECH-SOFTWARE/rtist-in-code
[3] Available at www.eclipse.dev/etrice
[4] Available at https://projects.eclipse.org/projects/modeling.papyrus-rt

5.2. Step 2: Identify Runtime Information

The collection of relevant runtime information is necessary to assess system performance and identify suitable adaptations. Which runtime information is most relevant depends on the application and its requirements and use cases. In the second step, the user identifies the runtime information necessary to assess system performance (and prior adaptation steps), determine the need for adaptation, and select the most suitable adaptation steps. The user then also ensures that this information is fed into the Adaptation Manager. Our approach considers two types of runtime information. For each type, the user needs to modify the model in different ways.

5.2.1. Internal Runtime Information

Internal runtime information is available on the model-level and includes, e.g., the time that a model component requires to respond to a request (*end-to-end delay*), the number of messages exchanged in a certain time interval, or the average size of the payload of certain messages. Our approach asks the user to develop suitable monitors for this information, i.e., components that collect the required information and feed it to the Adaptation Manager. For instance, to measure the end-to-end delay, i.e., the time between the receipt of an incoming request and the sending of the corresponding outgoing response, UML-RT allows the design of a monitor that starts a timer whenever a specific message (i.e., the request) arrives on a specific port of a specific component, stops the timer when the corresponding response message is sent out by the component, logs the elapsed time, and makes it available to the Adaptation Manager should it exceed a certain threshold. To facilitate the implementation of these monitors, our approach suggests to leverage model transformation as described in Step 3 below.

5.2.2. External Runtime Information

External runtime information is collected by the computing platform and includes the availability and utilization of different platform resources such as computing nodes, memory, and CPU. In our approach, the user must ensure that the required external runtime information is made available to the Adaptation Manager in the model. For instance, our prototype uses the Kubernetes Metrics Server to collect platform information and sends it to the model via a TCP/IP connection.

5.3. Step 3: Model Transformation

In general, model transformation is a process that entails the (in-place or out-place) conversion of a source model into a target model as directed by transformation rules, sometimes given

in a dedicated model transformation language such as ATL or Epsilon.

Our approach utilizes model transformation to facilitate the construction and integration of monitors (Step 2) but also to implement the partitioning of the model into stand-alone, independently executable, and deployable parts (Step 4).

For instance, to add a monitor to measure the end-to-end delay between a message (request) coming into a component and another message sent by the component in response, a model transformation can be used that takes a specification of the request and response messages and the component and port involved as input and creates the corresponding monitoring component (with, e.g., attributes to store start and end times and their differences) and integrates it appropriately into the existing model. The addition of other monitors to, e.g., count messages or determine the size of message data can be automated in a similar fashion. For instance, Figure 8 shows the structure of a model of our UAV simulation exemplar resulting from the integration of the monitors for measuring end-to-end delay, throughput, and data size.

The use of model transformation for partitioning the model is described in the next section. In our prototype, model transformation is accomplished using Kotlin and RTPoet (RTPoet 2021), our domain-specific library for UML-RT, which supports model creation and manipulation as well as code generation.

5.4. Step 4: Partitioning and Code and Script Generation

This step prepares the cloud deployment by partitioning the model, adding support for automatic failure recovery, and generating the necessary code and scripts. In our prototype, Step 4 is implemented with the help of KubeRT (KubeRT 2021), which serves as an automatic partitioning and code and script generation framework for C&C models.

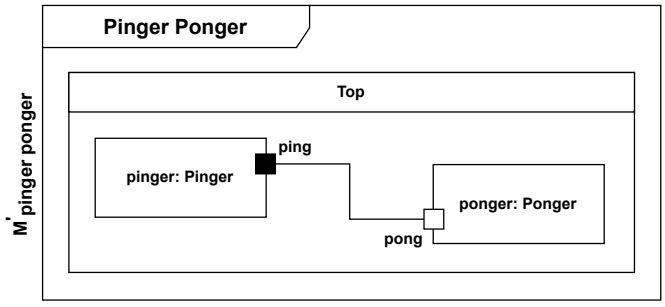

Figure 3 Pinger Ponger UML-RT model

5.4.1. Partitioning In the model created in Step 1 by the user, components represent separately executable and deployable units that can run on different nodes in the cloud environment and yet still communicate with each other as indicated by the connectors. In the context of the modeling language and MDE tool chosen in Step 1 to express the model of the system, the goal of the partitioning is to achieve this separate deployability, i.e., the partitioning turns every component in the model into its own separate model M'' such that the tool can be used to generate separately executable code for each component while fully preserving communication abilities to the components it is connected to.

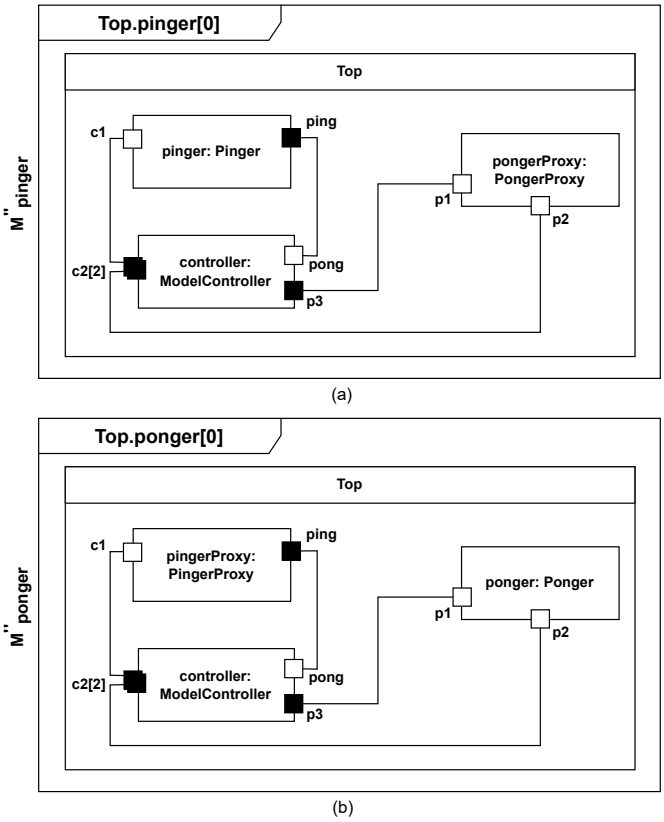

Figure 4 Pinger Ponger Model Generated by KubeRT

To achieve this partitioning, we use *proxy components*. Consider, for example, the model in Figure 3 containing two components `pinger` and `ponger` with a single connector. For this model, the partitioning process will produce two models as shown in Figure 4, one for `pinger` (Figure 4 (a)) and one for `ponger` (Figure 4 (b)). In the model for `pinger`, the component it is connected to in the original model (`ponger`) is replaced by a proxy component (`pongerProxy`). Similarly for the model created for `ponger`. Messages from `pinger` originally sent to `ponger` will now be sent to `pongerProxy` which will forward them (via remote communication) to the model that now represents `ponger`. Once they arrive at the model, the proxy representing `pinger` will forward it to the `ponger` component. Similarly for messages sent from `ponger`.

In our prototype, KubeRT realizes the partitioning with the help of model transformations. The proxy capsules use TCP by default to establish connections and messages are encoded in JSON format. Moreover, KubeRT is extensible and supports additional network protocols and data formats.

5.4.2. Support for Failure Recovery As part of the partitioning process, the model is also updated such that the resulting application can take advantage of the failure recovery capabilities that containerization management frameworks offer. For instance, Kubernetes can automatically restart failed containers. Our approach allows the modeled application to take advan-

tage of this capability. Concretely, it realizes a failure recovery process for individual components that guarantees that every message will be delivered and processed by its intended recipient assuming proper operating conditions are eventually restored.

To achieve this, as shown in Figure 4, a controller is added to every standalone model M'' created by the partitioning. The controller is placed between the components (`pinger` or `ponger`) and all proxies allowing it to intercept all messages exchanged between the components and the proxies. Moreover, the component is modified so that it can, upon request from the controller, serialize its current runtime state and send it to the controller (checkpointing), and recover from failure by restarting in a state sent to it by the controller (state recovery). Finally, an acknowledgment and retransmission protocol is added to the component so that it automatically retransmits messages that remain unacknowledged due to a failure. Before forwarding a message to the component, the controller obtains and saves the current runtime state of the component. Upon restart from a failure, the controller restarts the component with the previously stored state.

5.4.3. Code and Script Generation

To generate code from the model created in Step 1, the code generator of the MDE tool supporting the chosen modeling language is used. More precisely, for each of the models created by the partitioning step, separately executable code is created. The scripts and specification files necessary for building and deploying the code are generated as well.

In our prototype, KubeRT uses an existing UML-RT code generator (part of the open-source Eclipse Papyrus-RT tool (Papyrus-RT 2017)) to generate C++ code for each partitioned model along with the associated Docker image specification files, Kubernetes service specification files, Gradle build scripts, and Kubernetes deployment specification files to handle the deployment process. Concretely, for each generated model M'', KubeRT generates the following artifacts:

- **Docker file:** A Docker file to build a container image that executes the code generated from model M''. All images extend the same base image. The base image includes the toolchain required to build and execute UML-RT models.
- **Deployment YAML file:** A Kubernetes deployment specification file that requests the deployment of a Kubernetes pod running the image. The deployment YAML file is generated with its default deployment configurations, which can be modified as needed. For instance, Listing 1 shows the generated deployment YAML file for our UAV simulation exemplar.
- **Service YAML file:** A Kubernetes service specification file that exposes the TCP port of every proxy capsule in M''. Listing 2 shows the service YAML file for our UAV simulation exemplar.
- **Gradle build script:** A Gradle build script that covers the entire deployment process. The build script defines the set of tasks, and their dependencies, needed to deploy the model on the Kubernetes cluster.

5.5. Step 5: Automatic Containerization and Deployment

As the last step, the code and its corresponding dependencies, including Docker files, service YAML files, build scripts, and deployment YAML files are bundled together to generate Docker images of the application, which can be deployed on a local or a remote cluster. The resultant Docker images are subsequently published to the container registry of the cloud-native platform.

The generated application is deployed in a cloud-native environment using GCP. We created VM instances in the Google Compute Engine, which acts as the IaaS component of GCP. This approach ensured efficient scalability, effective resource management, and reliable performance within the cloud environment. We created a variable number of VMs and then set up Kubernetes. The local environment is configured to interact with the GCP and allows us to push, store, and manage project images in the GCP container registry. For remote Kubernetes deployment, we configured the Kubernetes dashboard and the command line tool *kubectl* and used them to manage the cluster and perform tasks such as monitoring, scaling, pod creation/deletion, and tracking resource usage across nodes.

6. Runtime View of Deployed Adaptive System

The runtime view of the deployed system is depicted in Figure 5, consisting of two primary sections: On-premises and Cloud-Native Infrastructure. The on-premises section has various frontend elements, including command line interfaces, Jupyter notebooks, and Kubernetes tools. In contrast, the cloud infrastructure is composed of a Virtual Private Cloud (VPC) firewall, virtual machines, and services responsible for hosting the Kubernetes cluster. Below, we describe these components and their role in our prototype in more detail.

6.1. On-premises

The on-premises environment allows the creation and configuration of the cloud environment that our application executes in, as well as observing and interacting with the application. The Google Cloud CLI allows users to configure credentials for interaction with the GCP and for configuring Docker such that, e.g., the images created by our Gradle scripts can be pushed to the GCP container registry.

Jupyter Notebooks allows the interaction with the GCP services and the remote Kubernetes cluster. In the context of our UAV exemplar, these notebooks are used for executing the behavioral strategies essential for model-level adaptation. They further include functions to process and analyze any data and logs generated by the application. Moreover, for experimental purposes, users can dynamically modify the platform at runtime through these notebooks (in addition to the automatic adaptation that the application may be carrying out).

The Kubectl command-line tool helps the user interact with the remote Kubernetes clusters running on the virtual machines. It allows users to, e.g., view logs, manage pods, edit deployments, and can help debug the deployed application.

The Kubernetes Dashboard is a web-based interactive interface for managing the Kubernetes cluster, providing real-time information on pods, services, nodes, and deployments, facili-

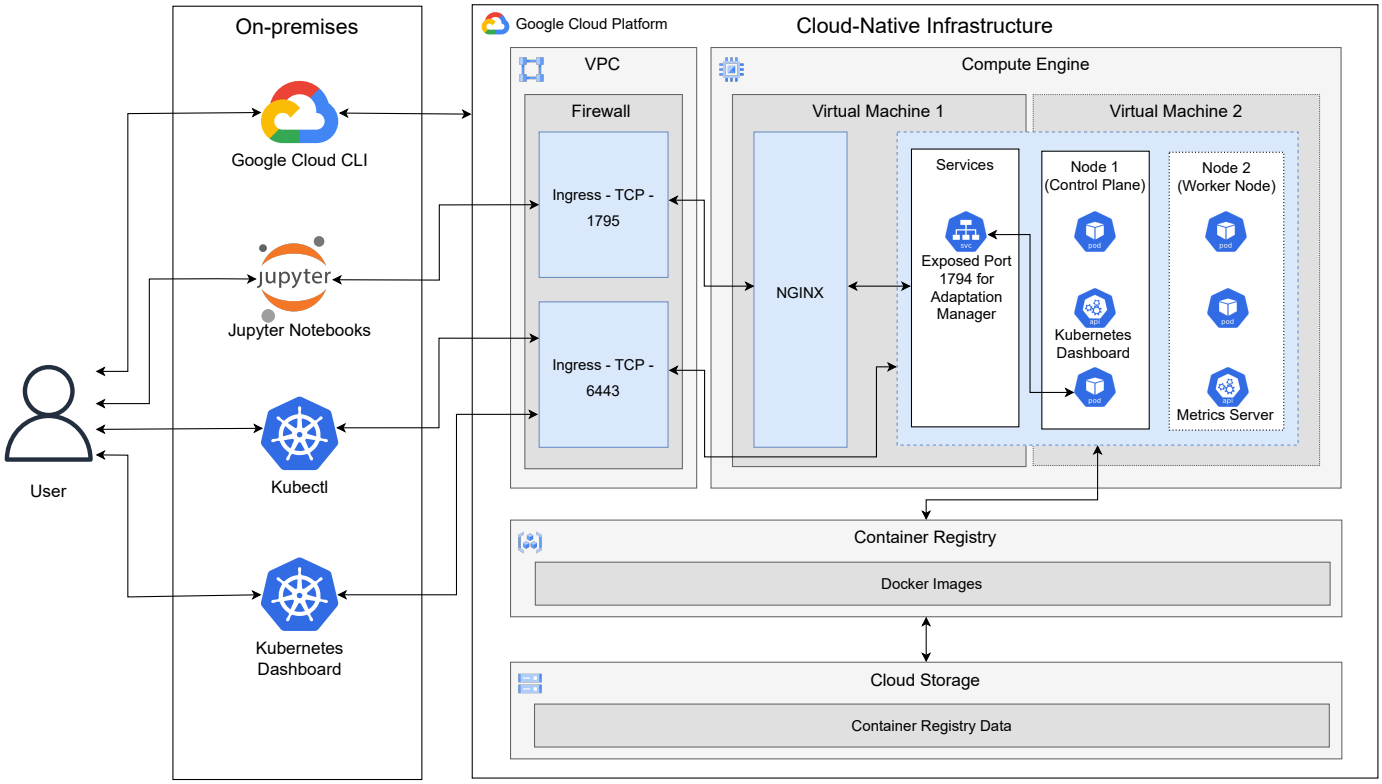

Figure 5 Runtime View of Deployed Adaptive System

tating cluster monitoring, resource management, and debugging. The dashboard is similar in functionality to Kubectl, but also supports the graphical display of information.

6.2. Cloud-Native Infrastructure

Our cloud-native infrastructure, hosted on GCP, incorporates various services that collectively allow the execution of our application.

The Virtual Private Cloud (VPC) service is used to define firewall rules. Two specific rules govern Ingress traffic on ports 1795 and 6443. Port 1795 is used by the application generated from the model to expose the TCP server which allows incoming traffic, while port 6443 enables the Kubernetes cluster services for the Kubectl application.

Another critical component is the Container Registry service, tasked with managing the Docker images generated by our toolchain. This service provides a reliable repository for storing these images. To further support image management, we utilize Cloud Storage, providing a secure and scalable solution for the long-term storage of Docker images.

The Compute Engine service is instrumental in creating virtual machines that serve as hosts for the Kubernetes cluster. At the core is a fundamental virtual machine functioning as the control plane node, which also hosts an NGINX server.

NGINX, a lightweight web server, provides web access to the application. Instead of directly exposing the application's port, NGINX offers high performance and scalability, efficiently forwarding remote connections to the relevant deployment within the Kubernetes cluster.

Within the Kubernetes environment, we deploy external services such as the Metrics Server and the Kubernetes Dashboard. Additionally, a service is created to expose TCP port 1794 on the deployment. This service allows the TCP server defined in the model to be exposed. The NGINX server forwards all TCP packets arriving on port 1795 to the exposed port 1794 of the deployment.

7. Evaluation

To evaluate our work and answer the research questions posed in Section 3, we have used our approach and prototype to create and deploy several cloud applications that all exhibit different kinds of self-adapting behavior.

7.1. Containerization and Deployment (RQ1)

The initial research question (RQ1) focuses on the ability to model, monitor, containerize, and deploy self-adaptive cloud applications. Below, we will describe four such applications that we have developed using our approach and prototype.

7.1.1. Word Count The problem is to count the number of occurrences of a specific word in a collection of text files. Using our prototype, we have created a model in which the managing component dynamically creates (instantiates) and invokes a counter component for each file. After all counter components have reported the number of occurrences of the work in each file, the manager computes the total number of occurrences. The manager uses an end-to-end delay monitor to track the amount of time required to determine the total count. Should this time

lie over some threshold, the manager dynamically triggers a platform adaptation that causes the application to be redeployed on a platform with more resources such as higher CPU or memory values. Containerization (Step 4) and deployment of the model (Step 5) took about 22 seconds and half a second, respectively (Table 1).

7.1.2. Sieve of Eratosthenes The Sieve application is similar to the Word Count application. The problem is to determine the prime numbers in a given integer interval. The manager dynamically creates a certain number of agents, assigns a subinterval to each of them, and then combines their results to obtain the overall result. Figure 6 shows a component diagram of the UML-RT model with four agents.

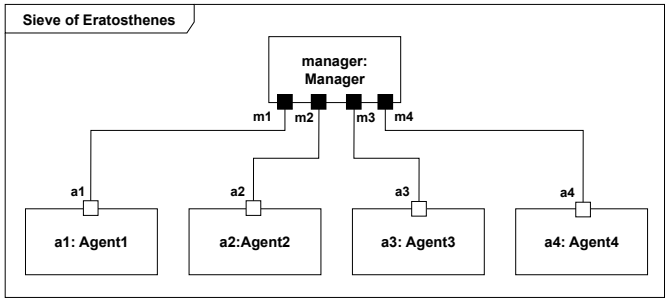

Figure 6 Sieve of Eratosthenes

As for the Word Count model, an end-to-end delay monitor is used to determine the time required, and the manager dynamically asks for more platform resources should this time be considered too high. Containerization and deployment of the Sieve model took slightly longer than for the Word Count model (Table 1).

7.1.3. Parcel Router A parcel router is an automated system that directs tagged parcels through a series of chutes and switchers to reach their respective destination bins. Based on parcel tags, switchers control door orientations to route parcels to the correct bins. The system is time-sensitive, and blockages can occur due to varying transit times through the chutes (Magee & Kramer 2006).

Using our approach and tooling we have created a self-adaptive, cloud-based simulation of a parcel router. As shown in Figure 7, the model contains a `gen` capsule that generates tagged parcels and three stages responsible for moving parcels to one of four bins. Each stage is further divided into chutes, switchers, and sensors (not shown in Figure 7).

When a parcel moves through the first chute, a sensor reads the parcel's tag and sends this information to the switcher. Upon receiving the parcel, the switcher can direct it to the right or left side, either to the next stage or one of the four bins. Not counting the monitors, the model contains a total of 21 (partially nested) components.

Delay and throughput monitors are used to, respectively, determine the time required for a parcel to reach its destination and to count the number of parcels delivered within a specific time interval. In contrast to the previous two models, the monitors in the Parcel Router model can be enabled and disabled

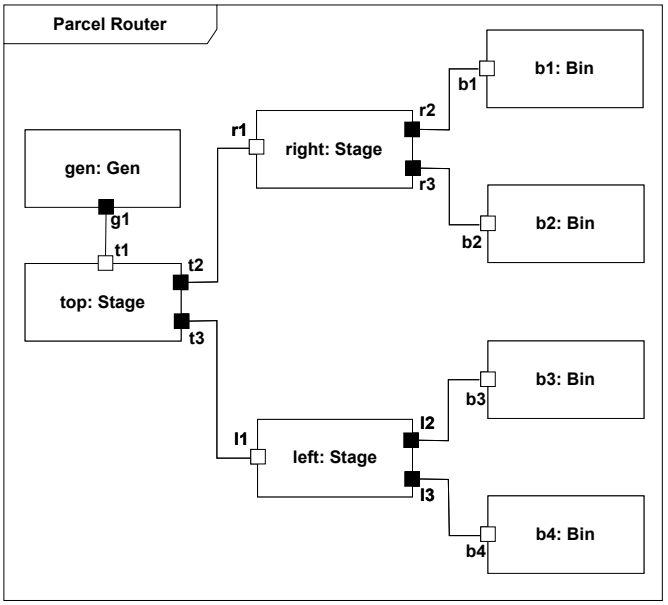

Figure 7 Parcel Router

dynamically by the manager using UML-RT's plug-in capsule mechanism. As before, platform adaptation is used to improve system performance at runtime. Containerization and deployment took just under a minute (58 secs) and just under a second (0.92 secs), respectively (Table 1).

7.1.4. UAV Exemplar The UAV exemplar emulates a group of UAVs navigating through an environment, identifying targets, and evading potential threats. Using our approach and tooling, we created a cloud-based simulation of UAVs with self-adaptive capabilities.

A component diagram of our model for this simulation is shown in Figure 8. The `DartMain` component is responsible for the main simulator functions, such as initializing the environment and simulating and interacting with the different sensors (for, e.g., targets and threats) and the UAVs. The `AdaptationMgr` realizes the strategies by assessing the need for strategy changes and instructing the relevant parts of the model to adopt the parameter values (discussed in Section 4.1) associated with the chosen strategy to improve target detection and threat avoidance. This iterative process continues until the simulation concludes or the UAVs are destroyed. The model includes delay, data size, and throughput monitors to track end-to-end delay, transmitted data size, and number of messages exchanged in a certain time interval. The `Metrics` component gathers system-level metrics, including CPU and memory usage, from the Kubernetes metric server. The `Log` component stores and manages logs from all capsules for each simulation run.

The UAV simulation takes various inputs such as map size, number of UAVs, and altitude level to initialize the environment. `AdaptationMgr` dispatches adaptation commands to `DartMain`, which executes them and reports back upon completion. The `Delay` monitor measures the time between the command being sent to `DartMain` and the component reporting back. If this time exceeds a threshold, `AdaptationMgr` initiates

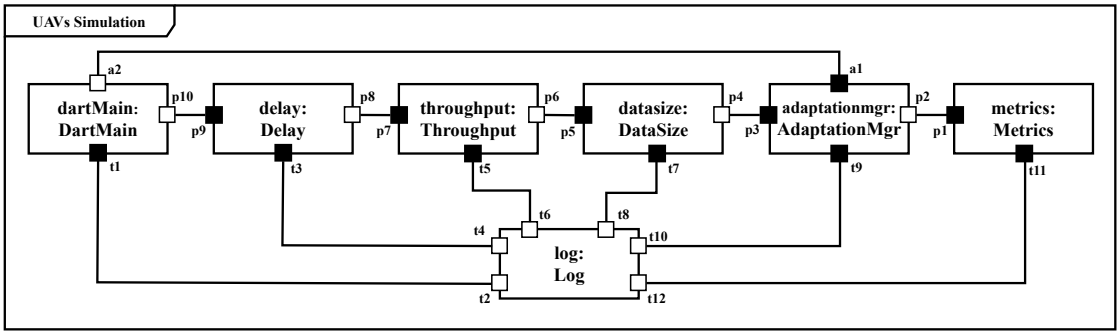

Figure 8 Model of the UAV Simulation after Integration of the Monitors

platform adaptation by requesting additional resources.

As discussed, the generated Gradle build script is responsible for managing the deployment process. It defines a series of tasks and their interdependencies required to deploy the model on a Kubernetes cluster. The `deploy` task utilizes the Kubernetes `kubectl` command-line tool to apply the deployment YAML and service YAML configuration files (Listings 1 and 2) on the cluster. To perform this, `deploy` relies on the `publish` task, which employs the Docker command-line tool to publish the container image generated by the `containerize` task. `Publish` pushes this image to an image registry, making it accessible to the Kubernetes cluster.

```yaml
# Deployment YAML File
apiVersion: apps/v1
kind: Deployment
metadata:
  name: adaptationmanager
  namespace: kubert
  labels:
    app: umlrt
    parent: "top"
    optional: "false"
spec:
  replicas: 1
  selector:
    matchLabels:
      name: adaptationmanager
  template:
    metadata:
      labels:
        name: adaptationmanager
        app: umlrt
    spec:
      volumes:
        - name: kubert
          persistentVolumeClaim:
            claimName: kubert
      containers:
        - name: adaptationmanager
          image: gcr.io/uavsimulation/adaptationmanager
          volumeMounts:
            - mountPath: "/mnt/data"
              name: kubert
          resources:
            requests:
              memory: "128Mi"
              cpu: "100m"
            limits:
              memory: "128Mi"
              cpu: "100m"
      terminationGracePeriodSeconds: 3
```

Listing 1 Kubernetes Deployment YAML Configuration

Since building the image requires the generated source code, the containerization is dependent on the `generate` task, which invokes the UML-RT C++ code generator on the model.

The deployment YAML file as shown in the Listing 1 is used to define and configure the Kubernetes deployment on a Kubernetes cluster. The Kubernetes resource definition consists of `apiVersion` and `kind` fields that specify the API version and the type of Kubernetes resource. The `metadata` section provides information about the deployment, including its name, namespace, and labels. Labels are used for resource identification and grouping. The `spec` section defines the desired characteristics of the deployment, with details such as the desired number of replica pods, selector criteria for pod control, and the pod template. The pod template, in turn, includes metadata with labels, specifications regarding volumes, a list of containers, their respective images, volume mounts, and resource requirements for CPU and memory. These specifications allow fine-tuning the resources allocated to pods within the Kubernetes cluster.

The service YAML file for the UAV model (Listing 2) is used to define a Kubernetes service in the `kubert` namespace. The `apiVersion` specifies the Kubernetes resource's API version and the `metadata` is for identification of the resource in the Kubernetes cluster. The `spec` section configures the specifications for a Kubernetes Service.

```yaml
# Service YAML File
apiVersion: v1
kind: Service
metadata:
  name: adaptationmanager
  namespace: kubert
  labels:
    app: umlrt
spec:
  selector:
    name: adaptationmanager
  ports:
    - name: top
      protocol: TCP
      port: 8000
      targetPort: 8000
```

Listing 2 Kubernetes Service YAML Configuration

It contains components such as the `selector`, which specifies the pods to which the `service` should route traffic based on their label. Additionally, the `ports` section defines the port settings for the service, including a port name, protocol, port number, and the target port within the selected pods. This configuration allows the service to effectively route and manage network traffic to the specified pods, enabling access to the service they provide on the port while using the TCP protocol.

The model for UAV simulation is significantly larger than the other models discussed in this section (about 50 components). As a result, containerization and deployment also took significantly longer (Table 1).

Table 1 Containerization and Deployment Time

Model	Containerization (sec)	Deployment (sec)
Word Count	22.47	0.53
Sieve of Eratosthenes	27.64	0.74
Parcel Router	57.52	0.92
UAV Simulation	232.96	2.67

The results in Table 1 demonstrate that the partitioning, containerization, and deployment of the models created in Steps 1 and 2 of our approach can be automated at a cost that is proportional to the size of the model and that is dominated by the containerization time. We note that our prototype focusses on demonstrating feasibility and that containerization and deployment times could probably be reduced via suitable optimization.

The answer to **RQ1** is that with our automatic containerization and deployment toolchain it is possible to partition the monolithic model into individual components and generate related configuration files which can be containerized and deployed, allowing the model-driven development of containerized applications deployable on local and cloud-native infrastructure.

7.2. Failure Recovery (RQ2)

To evaluate the effectiveness of the automatic failure recovery capability added to the model in Step 4 of the approach, we used the parcel router model. While executing the model on a Kubernetes cluster, an external Python script was used to induce failures. At random times, the script selects a random subset of the 21 components and injects a command that causes a kernel panic into their containers. In total, 67 failures were induced, each with an average of 8 components. During the execution, the maximum number of simultaneously failed components hit 21. Nonetheless, our failure recovery mechanism managed to recover from every failure, and all the parcels were eventually routed to their correct destination. Figure 9 shows the time taken to recover from failures and resume the execution for a varying number of failed components. The majority of time is taken by the Kubernetes engine to restart the failed container while the remaining time is taken by the state and message recovery process.

As described in Section 5.4.2, our approach requires serializing and saving the state of a component whenever it has fully processed a message. Our experiments showed that, on average, this checkpointing process imposed a 3.2% overhead on the message processing time, which can be a concern for applications with a high message frequency. The overhead can also be more significant for components with a large state. Further improvements could be made by, e.g., incrementally saving the state by computing deltas.

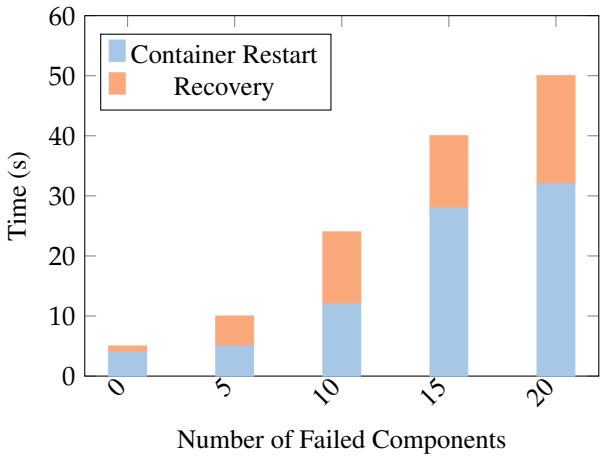

Figure 9 Performance of the Failure Recovery Process

The answer to **RQ2** is that, building on Kubernetes' failure recovery capabilities, relatively strong fault tolerance guarantees can be offered on the model-level and implemented through mechanisms (checkpointing, state recovery, and retransmission) that are added automatically through model transformation and remain transparent to the user.

7.3. Model-level Adaptation (RQ3a)

To evaluate the ability of the approach and prototype to adapt to changing user requirements using adaptation on the model-level, we use our UAV exemplar and allow users to change the type of the mission (surveillance, attack, or balanced) at runtime. As shown in Figure 10, this user input is sent to the Adaptation Manager and possibly causes it to change the strategy that governs the choice of the parameters that influence the behavior of the UAVs. The manager then (re-)initializes the Kubernetes Metric Server used to collect platform-level metrics (i.e., external runtime information) and then runs the simulation using the chosen strategy. During the simulation, internal and external runtime information is sent to the Adaptation Manager and the Log component from, respectively, the delay, throughput, data size monitors, and the metric server (Figure 8). Upon completion of the simulation, results are collected and passed to the user together with the system log.

Table 2 Effectiveness of Strategies

	Strategy		
Metric	Conservative	Aggressive	Balanced
ADP	18.6	11.7	14.2
ANTF	1.2	2.5	1.8
AMSF	0.1	0	0.5

To evaluate the effectiveness of the strategy selection and change process, we use the metrics described in Section 4 to assess different missions using different strategies. The results are shown in Table 2. As we can see, the average destruction position (ADP) is highest for the conservative strategy, least for the aggressive strategy, and moderate for the balanced strategy.

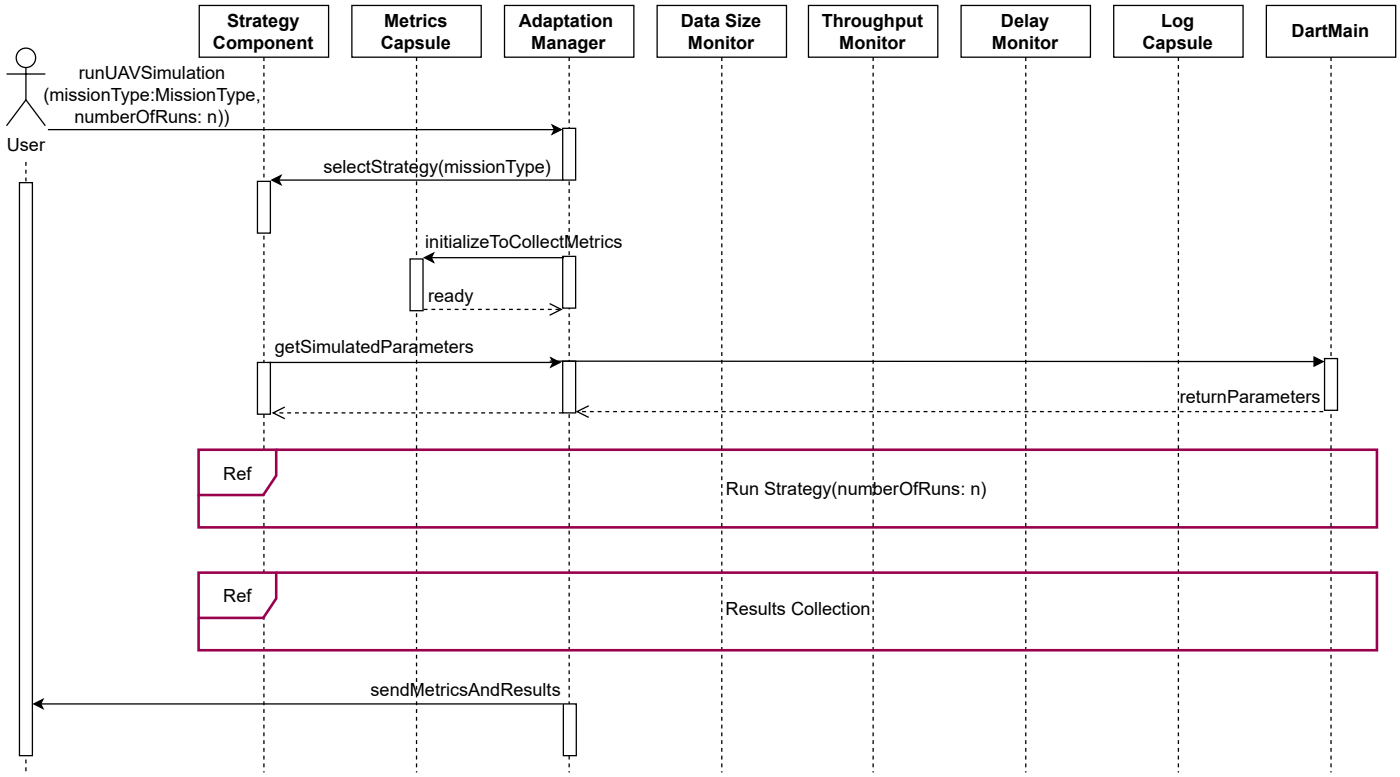

Figure 10 Sequence Diagram of UAV Simulation

The aggressive strategy found the highest average number of targets (ANTF), followed by the balanced strategy, and the conservative strategy found the least. The balanced strategy has the highest average mission success factor (AMSF), followed by the conservative strategy, while the aggressive strategy has the lowest. We observe that the strategies not only impact UAV performance, but also cause missions to support the mission type chosen by the user, i.e., the user-level requirements. The strategy selection process takes place at runtime and thus allows the simulation to adapt effectively to dynamic changes to user-level requirements.

The response to **RQ3a** is that our approach and prototype tool can support the model-driven development of cloud applications that perform model-level adaptation to effectively respond to application requirements that change at runtime.

7.4. Platform-level Adaptation (RQ3b)

To evaluate the ability of the approach and prototype to adapt to changing user requirements via changes to the computing environment we again use the UAV exemplar. We define two types of platforms: minimal and maximal. A *minimal platform* offers only modest resources, but would be less expensive. Concretely, a minimal platform offers 100 milliCPU (units of virtual CPU in Kubernetes), 128 MebiBytes (units of virtual memory in Kubernetes), and one node. For instance, the Kubernetes deployment configuration file in Listing 1 would run the UAV simulation on a minimal platform. Conversely, a *maximal platform* offers more resources but would also be more expensive. Concretely, it offers 500 milliCPU, 512 MebiBytes, and a cluster of three nodes.

We enable the Adaptation Manager to dynamically change the deployment configuration file (i.e., Listing 1) and request a different platform, if the runtime information it has access to suggests that the current platform is not suitable. For instance, a large end-to-end delay between the manager issuing a command to the simulation (i.e., the `DartMain` component) and the response might suggest that the application is under-resourced and would prompt the manager to switch from a minimal to a maximal platform.

Table 3 Effect of Change in Resources on Performance

Metric	Strategy	Platform	Values	% Gain
ADP	Conservative	Minimum	18.6	-
ADP	Conservative	Maximum	24.2	≈ 30%
ANTF	Aggressive	Minimum	2.5	-
ANTF	Aggressive	Maximum	3.8	≈ 50%
AMSF	Balanced	Minimum	0.5	-
AMSF	Balanced	Maximum	0.75	≈ 50%

We then determine the effect of the platform change on the mission metrics. For instance, Table 3 shows the impact of replacing a minimum platform by a maximum platform. We see that in the context of the conservative strategy the availability of more resources leads to an improvement of the average destruction position (ADP) of about 30%. For the aggressive strategy, the average number of targets found (ANTF) increases by 50%. For the balanced strategy, the average mission success

factor (AMSF) also increases by 50%.

In other words, we see evidence that the availability of more resources does indeed allow the application to satisfy the user requirements (i.e., the chosen mission type) to a higher degree. Intuitively, this is because relevant parts of the application become more responsive and are able to process inputs or respond to requests or changing runtime information such as sensor input with less delay, allowing the application to, e.g., detect a target or threat where previously it did not.

In response to **RQ3b**, we conclude that our approach is able to support the model-driven development of cloud applications that use dynamic platform adaptation effectively to respond to changing user requirements.

8. Discussion

8.1. Choice of Modeling Language

Our approach advocates the use of actor-based C&C architectural description languages for the modeling of self-adaptive, stateful cloud applications. In the context of UML-RT, our prototype puts some of the necessary techniques and tools in place to accomplish this. Although originally designed for real-time embedded systems, we believe that UML-RT does allow the effective, high-level description of the structure and behavior of self-adaptive cloud applications, and that it can accommodate current best architectural practices for the design of self-adaptive systems such as the MAPE-K reference architecture and its variants. It can be used to collect relevant runtime information to determine the need for adaptation, which can then be realized using built-in, model-level mechanisms that modify the structure and behavior of the system dynamically (Kahani et al. 2017). Existing tooling for UML-RT such as Papyrus-RT greatly facilitated the development of our prototype.

On the more negative side, UML-RT uses state machines exclusively for the description of behavior which might represent an obstacle for some users. However, emerging support for purely textual and graphical/textual-hybrid representations complements traditional graphical representations and might help lower the barrier to entry for some users.

There are several languages that also follow the actor model and that could be used in place of UML-RT state machines to describe the behavior of components. Candidates include Akka, Erlang (and its extensions such as Elixir), and Orleans and the reactive programming model that they offer[5].

9. Conclusion and Future Work

We have presented an approach for the model-driven development of self-adaptive cloud applications, together with supporting tooling. The approach uses C&C models based on the actor paradigm and integrates model- and platform-level adaptation. Approach and tooling were evaluated using several examples, including an exemplar involving the simulation of a collection of UAVs. Results suggest that the approach is feasible and can support effective self-adaptation.

[5] github.com/akka, github.com/topics/erlang, github.com/dotnet/orleans, www.reactivemanifesto.org

Our ongoing and future work focuses on the items below.

1. Further leveraging of capabilities of container management platforms: Some of the orchestration mechanisms for, e.g., automatic scaling assume stateless applications, complicating their use for the stateful applications. Our approach shows how this challenge can be dealt with in the context of failure recovery by enabling components to recover the previous state. However, it is unclear to what extent existing platform support for automatic scaling of stateful applications (such as StatefulSets in Kubernetes) can be used to extend our approach and allow the platform to dynamically replicate certain components in response to increased demand.

2. Refine modeling part of approach: Modeling the application can be a big task, and it would be useful to facilitate Step 1 of our approach. Support for architectural patterns and reusable components on the model-level would help. For instance, UML-RT supports component reuse and specialization via inheritance and component libraries providing a possible avenue to, e.g., further facilitate the definition of monitors and adaptation managers in the context of our prototype.

3. Refine adaptation support: A related avenue for future work is support for more complex, tradeoff-aware decision-making by, e.g., integrating platform costs and allowing the user to specify an optimization function, i.e., how different performance and cost aspects should be weighed. A considerable amount of work on this topic already exists, but the realization in the context of our approach needs to be explored.

4. Additional tool support: Many tools exist to support the collection, analysis, visualization, and storage of data. Open-source tools such as Prometheus and Grafana seem particularly relevant. Allowing these tools to be integrated into our toolchain might help enhance its user-friendliness and capabilities.

References

Agha, G. A. (1985). *Actors: A model of concurrent computation in distributed systems.* (Tech. Rep.). Massachusetts Institute of Technology, Cambridge Artificial Intelligence Lab.

Alfonso, I., Garcés, K., Castro, H., & Cabot, J. (2021). Modeling self-adaptative IoT architectures. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)* (pp. 761–766).

Bagherzadeh, M., Jahed, K., Combemale, B., & Dingel, J. (2021). Live modeling in the context of state machine models and code generation. *Software and Systems Modeling*, *20*(1), 795-819.

Bencomo, N., Goetz, S., & Song, H. (2019). Models@run.time: A guided tour of the state of the art and research challenges. *Software and Systems Modeling*, *18*, 3049–3082.

Beyer, B., Jones, C., Murphy, N., & Petoff, J. (2016). *Site reliability engineering: How Google runs production systems*. O'Reilly.

Bradbury, J. S., Cordy, J. R., Dingel, J., & Wermelinger, M. (2004). A survey of self-management in dynamic software architecture specifications. In *1st ACM SIGSOFT Workshop on Self-managed Systems* (p. 28-33).

Casalicchio, E. (2019). Container orchestration: A survey. *Systems Modeling: Methodologies and Tools*, 221–235.

Cheng, S.-W., & Garlan, D. (2012). Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software*, *85*(12), 2860–2875.

Elkhodary, A., Esfahani, N., & Malek, S. (2010). FUSION: A framework for engineering self-tuning self-adaptive software systems. In *18th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 7–16).

Epifani, I., Ghezzi, C., Mirandola, R., & Tamburrelli, G. (2009). Model evolution by run-time parameter adaptation. In *31st International Conference on Software Engineering* (pp. 111–121).

Farokhi, S., Jamshidi, P., Brandic, I., & Elmroth, E. (2015). Self-adaptation challenges for cloud-based applications: A control theoretic perspective. In *10th International Workshop on Feedback Computing* (Vol. 2015).

Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B., & Steenkiste, P. (2004). Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, *37*(10), 46–54.

Herbst, N. R., Kounev, S., & Reussner, R. (2013). Elasticity in cloud computing: What it is, and what it is not. In *10th International Conference on Autonomic Computing*.

Hummaida, A. R., Paton, N. W., & Sakellariou, R. (2016). Adaptation in cloud resource configuration: a survey. *Journal of Cloud Computing*, *5*, 1–16.

Kahani, N., Hili, N., Cordy, J. R., & Dingel, J. (2017). Evaluation of UML-RT and Papyrus-RT for modelling self-adaptive systems. In *IEEE/ACM 9th International Workshop on Modelling in Software Engineering* (pp. 12–18).

Kephart, J. O., & Chess, D. M. (2003). The vision of autonomic computing. *Computer*, *36*(1), 41-50.

Kramer, J., & Magee, J. (2007). Self-managed systems: An architectural challenge. In *Future of Software Engineering* (pp. 259–268).

Krupitzer, C., Roth, F. M., VanSyckel, S., Schiele, G., & Becker, C. (2015). A survey on engineering approaches for self-adaptive systems. *Pervasive and Mobile Computing*, *17*, 184–206.

KubeRT - Automated partitioning and deployment of UML-RT models. (2021). https://github.com/qumase/kubert. (Accessed: 2023-11-27)

Magee, J., & Kramer, J. (2006). *Concurrency: State models & Java programs (2nd ed.)*. Wiley.

McDirmid, S. (2016). The promise of live programming. In *2nd International Workshop on Live Programming (LIVE'16)*.

Mell, P., & Grance, T. (2011). The NIST definition of cloud computing. National Information Technology Laboratory.

Mohammed, M. M. (2022). Dynamic adaptation for distributed systems in model-driven engineering. In *25th International Conference on Model-Driven Engineering Languages and Systems: Companion Proceedings* (pp. 146–151).

Moreno, G., Kinneer, C., Pandey, A., & Garlan, D. (2019). DARTSim: An exemplar for evaluation and comparison of self-adaptation approaches for smart cyber-physical systems. In *14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)* (pp. 181–187).

Nakagawa, H., Ohsuga, A., & Honiden, S. (2012). Towards dynamic evolution of self-adaptive systems based on dynamic updating of control loops. In *6th International Conference on Self-Adaptive and Self-Organizing Systems* (pp. 59–68).

Oquendo, F. (2004). π-ADL: An architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. *ACM SIGSOFT Software Engineering Notes*, *29*(3), 1–14.

Papyrus-RT - A domain-specific modelling tool. (2017). https://www.eclipse.org/papyrus-rt/. (Accessed: 2021-01-20)

Porter, B., Filho, R. R., & Dean, P. (2020). A survey of methodology in self-adaptive systems research. In *IEEE International Conference on Autonomic Computing and Self-Organizing Systems*.

Pueschel, G., Goetz, S., Wilke, C., & Assmann, U. (2013). Towards systematic model-based testing of self-adaptive software. In *5th International Conference on Adaptive and Self-Adaptive Systems and Applications*.

RTPoet-DSL - UML-RT based domain-specific language. (2021). https://github.com/qumase/rtpoet-dsl. (Accessed: 2023-11-27)

RTPoet-UML-RT based domain-specific library. (2021). https://github.com/qumase/rtpoet. (Accessed: 2023-11-27)

Salehie, M., & Tahvildari, L. (2009). Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, *4*(2), 1–42.

Saxena, T., Dubey, A., Balasubramanian, D., & Karsai, G. (2010). Enabling self-management by using model-based design space exploration. In *7th IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems*.

Selic, B. (2022). Modeling of real-time software systems. In *Handbook of Real-Time Computing* (pp. 25–98). Springer.

Selic, B., Gullekson, G., & Ward, P. (1994). *Real-time object-oriented modeling*. John Wiley & Sons.

Spyker, A. (2020). Disenchantment: Netflix titus, its feisty team, and daemons. In *InfoQ*. (www.infoq.com/presentations/netflix-titus-2018)

Tajalli, H., Garcia, J., Edwards, G., & Medvidovic, N. (2010). PLASMA: A plan-based layered architecture for software model-driven adaptation. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering* (pp. 467–476).

van Rozen, R., & van der Storm, T. (2019). Toward live domain-specific languages. *Software and Systems Modeling*, *18*(1), 195-212.

Vogel, T., & Giese, H. (2014). Model-driven engineering of self-adaptive software with EUREMA. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, *8*(4), 1–33.

Weyns, D. (2020). *An introduction to self-adaptive systems: A contemporary software engineering perspective*. Wiley.

Weyns, D., Gerostathopoulos, I., Abbas, N., Andersson, J., Biffl, S., Brada, P., ... Pelliccione, P. (2023). Self-adaptation in industry: A survey. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*.

About the authors

Mufasir Muthaher Mohammed is a doctoral researcher at the School of Computing at Queen's University. He received his master's degree in Computer Science from Blekinge Institute of Technology, Sweden. His primary research interests include model-driven engineering, autonomic computing, cyber security, and high-performance computing. You can contact him at mufasirmuthaher.mohammed@queensu.ca.

Karim Jahed is a senior software engineer at Cisco Systems and an adjunct instructor at Queen's University. He received his Ph.D. in Computing from Queen's University. His current research interests include model-driven development, distributed systems, and IoT. You can contact him at kj38@queensu.ca.

Reza Ahmadi is a post-doctoral researcher at École de Technologie Supérieure at Université du Québec. He received his Ph.D. in Computing from Queen's University. His research interests include model-driven engineering, software testing, and DevOps. You can contact him at reza.ahmadi@etsmtl.ca.

Juergen Dingel is a professor at the School of Computing at Queen's University, where he leads the Modeling and Analysis in Software Engineering (MASE) research group. He received his Ph.D. in Computer Science from Carnegie Mellon University. His research interests include software modeling, model-driven engineering, formal methods, and software quality assurance. You can contact him at dingel@queensu.ca or visit https://research.cs.queensu.ca/home/dingel/.

David Lamb is an associate professor at the School of Computing at Queen's University. He received his Ph.D. in Computer Science from Carnegie Mellon University. His research interests include software design methodologies and software architecture. You can contact him at dal@queensu.ca or visit https://www.cs.queensu.ca/people/David/Lamb.